Lecture 17: Classes

Dr. Mohammed Hawa
Electrical Engineering Department
University of Jordan

Classes

- C++ classes allow us to define totally new data types.
- The new data types can have both data and the functions operating on such data close to each other.
- A class usually has a constructor and destructor.
- *Examples*:
  - Complex class with operators on complex numbers
  - String class with operators on strings

Complex Class

```
#include <iostream>

class ComplexNumber {
private:
   double real;
   double imaginary;

public:
   ComplexNumber(double real = 0.0, double imag = 0.0);

   double Norm();
   double Arg();

   // friend declaration grants access to private and protected members of
   // the class where the friend declaration appears
   friend ComplexNumber operator+ (const ComplexNumber &a, const ComplexNumber &b);
   friend ComplexNumber operator- (const ComplexNumber &a, const ComplexNumber &b);
   friend ComplexNumber operator* (const ComplexNumber &a, const ComplexNumber &b);
   friend ComplexNumber operator/ (const ComplexNumber &a, const ComplexNumber &b);

   friend std::ostream& operator<< (std::ostream &out, const ComplexNumber &c);
};
```


Constructor and Functions

```
ComplexNumber::ComplexNumber(double rr, double ii)
       : real(rr), imaginary(ii)
{
}

double ComplexNumber::Norm()
{
       return sqrt(real*real + imaginary*imaginary); // scalar
}

double ComplexNumber::Arg()
{
       return std::atan2(imaginary, real); // in radians
}
```


Operator Overload

```
ComplexNumber operator+ (const ComplexNumber &a, const ComplexNumber &b)
{
	ComplexNumber result;

	result.real = a.real + b.real;
	result.imaginary = a.imaginary + b.imaginary;

	return result;
}

ComplexNumber operator- (const ComplexNumber &a, const ComplexNumber &b)
{
	ComplexNumber result;

	result.real = a.real - b.real;
	result.imaginary = a.imaginary - b.imaginary;

	return result;
}
```



```
ComplexNumber operator* (const ComplexNumber &a, const ComplexNumber &b)
{
	ComplexNumber result;

	result.real = (a.real * b.real - a.imaginary * b.imaginary);
	result.imaginary = (a.real * b.imaginary + a.imaginary * b.real);

	return result;
}

ComplexNumber operator/ (const ComplexNumber &a, const ComplexNumber &b)
{
	ComplexNumber result;
	double magnitude_b = b.real * b.real + b.imaginary * b.imaginary;

	result.real = (a.real * b.real + a.imaginary * b.imaginary) / magnitude_b;
	result.imaginary = (a.imaginary * b.real - a.real * b.imaginary) / magnitude_b;

	return result;
}

std::ostream& operator<< (std::ostream &out, const ComplexNumber &c)
{
	out << "(" << c.real << " + j " << c.imaginary << ")";
	return out;
}
```

Test

```
ComplexNumber x(1, 2), y(3, 0), z;
z = ComplexNumber(3, 0);

std::cout << "x is: " << x << std::endl;
std::cout << "y is: " << y << std::endl;
std::cout << "z is: " << z << std::endl;

ComplexNumber sum        = x + y;
ComplexNumber difference = x - y;
ComplexNumber product    = x * y;
ComplexNumber quotient   = sum / z;

std::cout << "sum (x + y) is: " << sum << std::endl;
std::cout << "difference (x - y) is: " << difference<< std::endl;
std::cout << "product (x * y) is: " << x * y << std::endl;
std::cout << "quotient (sum / z) is: " << quotient << std::endl;

std::cout << "The magnitude of x is:  " << x.Norm() << std::endl;
std::cout << "The phase of x is:  " << x.Arg() << std::endl;
```

```
x is: (1 + j 2)
y is: (3 + j 0)
z is: (3 + j 0)
sum (x + y) is: (4 + j 2)
difference (x - y) is: (-2 + j 2)
product (x * y) is: (3 + j 6)
quotient (sum / z) is: (1.33333 + j 0.666667)
The magnitude of x is:  2.23607
The phase of x is:  1.10715
```


String class

```
#include <stdlib.h> // malloc, realloc
#include <string> // strlen
#include <stdexcept> // out_of_range

// The character are stored in a char buffer, using malloc rather than new,
// which makes it possible to use realloc.
class myString {
	char* p; // appended with null

public:
	myString(const char* s);
	myString(const myString&);
	myString();
	virtual ~myString();
	void init(const char* s);
	myString& operator=(const char*);
	myString& operator=(const myString&);
	myString& operator+=(const myString&);
	friend myString operator+(const myString& lhs, const myString& rhs);
	char operator[](const size_t n) const;
	bool operator==(const char*) const;
	bool operator==(const myString&) const;
	myString substr(const size_t start, const size_t length) const;
	size_t size() const;
	const char* c_str() const;
	void clear();
};
```


Constructor / Destructor

```
myString::myString(const char* s)
{
	init(s);
}

myString::myString(const myString& s)
{
	init(s.p);
}

myString::myString()
{
	init(""); // empty string
}

myString::~myString()
{
	if(p != NULL)
		free(p);
}

void myString::init(const char* s)
{
	const size_t length = strlen(s);
	p = static_cast<char*>(malloc(length+1)); // for NULL

	if( p == NULL )
		throw std::bad_alloc("myString::myString");
		// fail to allocate (p remains NULL)
	else
		memcpy(p, s, length); // including NULL

	p[length] = '\0'; // terminate by NULL
}
```


Operator Overload

```
myString& myString::operator= (const char* s)
{
	if ( p != s )
	{
		if(p != NULL)
			free(p);

		init(s);
	}

	return *this;
}

myString& myString::operator= (const myString& s)
{
	return operator= (s.p);
}
```

Operator Overload

```
myString& myString::operator+= (const myString& s)
{
	const size_t length_p = strlen(this->p);
	const size_t length_s = strlen(s.p) + 1; // for one NULL
	this->p = static_cast<char*>(realloc(this->p, length_p + length_s));

	if( p == NULL )
		return *this; // fail to allocate (p remains NULL)
	else
		memmove(p+length_p, s.p, length_s); // p and s.p MAY overlap

	return *this;
}

myString operator+ (const myString& lhs, const myString& rhs)
{
	return myString(lhs) += rhs;
}

// checked access
char myString::operator[] (const size_t n) const
{
	if ( n > strlen(p) )
		throw std::out_of_range("myString::operator[]");
	return p[n];
}
```


Operator Overload

```
bool myString::operator== (const char* s) const
{
	return !strcmp(p, s);
}

bool myString::operator== (const myString& s) const
{
	return !strcmp(p, s.p);
}

myString myString::substr(const size_t start, size_t length) const
{
	size_t fulllength = strlen(p);

	if ( start > fulllength || length <= 0 || (start+length) >= fulllength)
		throw std::out_of_range("myString::substr");

	myString s;
	free(s.p);

	s.p = static_cast<char*>(malloc(length+1));
	if( s.p == NULL )
		return s; // fail to allocate (p remains NULL)
	else
		memcpy(s.p, p + start, length); // including NULL

	s.p[length] = '\0'; // NULL
	return s;
}
```

More

```
size_t myString::size() const
{
	return strlen(p);
}

const char* myString::c_str() const
{
	return p; // NULL-terminated character array
}

void myString::clear()
{
	free(p);
	init("");
}
```


Test

```
myString h("Hello");
myString s(" ");
myString w("World");

std::cout << h.c_str() << std::endl;

h += (s + w);

std::cout << h.c_str() << std::endl;
std::cout << s.c_str() << std::endl;
std::cout << w.c_str() << std::endl;

myString test = h.substr(0, 2);

std::cout << test.c_str() << std::endl;
std::cout << test.size() << std::endl;

h.clear();
std::cout << h.c_str() << std::endl;

// should throw out_of_range
try {
	w.substr(2, 10);
}
catch(const std::out_of_range& e) {
	std::cout << "Exception was thrown: " << e.what() << std::endl;
}
```

```
Hello
Hello World

World
He
2

Exception was thrown: myString::substr
Press any key to continue . . .
```